Data Analysis with PANDAS

CHEAT SHEET

Created By: Arianne Colton and Sean Chen

Data Structures

Series (1D)

One-dimensional array-like object containing an array of data (of any **NumPy** data type) and an associated array of data labels, called its **"index"**. If index of data is not specified, then a default one consisting of the integers 0 through N-1 is created.

Create Series	`series1 = pd.Series ([1, 2], index = ['a', 'b'])` `series1 = pd.Series(dict1)*`
Get Series Values	`series1.values`
Get Values by Index	`series1['a']` `series1[['b','a']]`
Get Series Index	`series1.index`
Get Name Attribute (None is default)	`series1.name` `series1.index.name`
** Common Index Values are Added	`series1 + series2`
Unique But Unsorted	`series2 = series1.unique()`

* Can think of Series as a fixed-length, ordered dict. Series can be substitued into many functions that expect a dict.

** Auto-align differently-indexed data in arithmetic operations

DataFrame (2D)

Tabular data structure with ordered collections of columns, each of which can be different value type.

Data Frame (DF) can be thought of as a dict of Series.

Create DF
(from a dict of equal-length lists or NumPy arrays)
	`dict1 = {'state': ['Ohio', 'CA'], 'year': [2000, 2010]}`
	`df1 = pd.DataFrame(dict1)`
	# columns are placed in sorted order
	`df1 = pd.DataFrame(dict1, index = ['row1', 'row2'])`
	# specifying index
	`df1 = pd.DataFrame(dict1, columns = ['year', 'state'])`
	# columns are placed in your given order

* Create DF (from nested dict of dicts) The inner keys as row indices	`dict1 = {'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 3, 'row2': 4} }` `df1 = pd.DataFrame(dict1)`

Get Columns and Row Names	`df1.columns` `df1.index`
Get Name Attribute (None is default)	`df1.columns.name` `df1.index.name`
Get Values	`df1.values` # returns the data as a 2D ndarray, the dtype will be chosen to accomandate all of the columns
** Get Column as Series	`df1['state'] or df1.state`
** Get Row as Series	`df1.ix['row2'] or df1.ix[1]`
Assign a column that doesn't exist will create a new column	`df1['eastern'] = df1.state == 'Ohio'`
Delete a column	`del df1['eastern']`
Switch Columns and Rows	`df1.T`

* Dicts of Series are treated the same as Nested dict of dicts.

** Data returned is a 'view' on the underlying data, NOT a copy. Thus, any in-place modificatons to the data will be reflected in df1.

Panel Data (3D)

Create Panel Data : (Each item in the Panel is a DF)

```
import pandas_datareader.data as web
panel1 = pd.Panel({stk : web.get_data_
yahoo(stk, '1/1/2000', '1/1/2010')
for stk in ['AAPL', 'IBM']})
# panel1 Dimensions : 2 (item) * 861 (major) * 6 (minor)
```

"Stacked" DF form : (Useful way to represent panel data)

```
panel1 = panel1.swapaxes('item', 'minor')
panel1.ix[:, '6/1/2003', :].to_frame() *
```

=> Stacked DF (with **hierarchical indexing** **) :

		Open	High	Low	Close	Volume	Adj-Close
major	minor						
2003-06-01	AAPL						
	IBM						
2003-06-02	AAPL						
	IBM						

Data Structures continued

* DF has a "to_panel()" method which is the inverse of "to_frame()".

** Hierarchical indexing makes N-dimensional arrays unnecessary in a lot of cases. Aka prefer to use Stacked DF, not Panel data.

Index Objects

Immutable objects that hold the axis labels and other metadata (i.e. axis name)

- i.e. Index, MultiIndex, DatetimeIndex, PeriodIndex
- Any sequence of labels used when constructing Series or DF internally converted to an Index.
- Can functions as fixed-size set in additional to being array-like.

Hierarchical Indexing

Multiple index levels on an axis : A way to work with higher dimensional data in a lower dimensional form.

MultiIndex :

```
series1 = Series(np.random.randn(6), index =
[['a', 'a', 'a', 'b', 'b', 'b'], [1, 2, 3,
1, 2, 3]])
series1.index.names = ['key1', 'key2']
```

Series Partial Indexing	`series1['b']` # Outer Level `series1[:, 2]` # Inner Level

DF Partial Indexing	`df1['outerCol3','InnerCol2']`
	Or
	`df1['outerCol3']['InnerCol2']`

Swaping and Sorting Levels

Swap Level (level interchanged) *	`swapSeries1 = series1.swaplevel('key1', 'key2')`
Sort Level	`series1.sortlevel(1)` # sorts according to first inner level
Common Ops : Swap and Sort **	`series1.swaplevel(0, 1).sortlevel(0)` # the order of rows also change

* The order of the rows do not change. Only the two levels got swapped.

** Data selection performance is much better if the index is sorted starting with the outermost level, as a result of calling `sortlevel(0)` or `sort_index()`.

Summary Statistics by Level

Most stats functions in DF or Series have a "level" option that you can specify the level you want on an axis.

Sum rows (that have same 'key2' value)	`df1.sum(level = 'key2')`
Sum columns ..	`df1.sum(level = 'col3', axis = 1)`

- Under the hood, the functionality provided here utilizes panda's "**groupby**".

DataFrame's Columns as Indexes

DF's "set_index" will create a new DF using one or more of its columns as the index.

New DF using columns as index	`df2 = df1.set_index(['col3', 'col4']) * ‡` # col3 becomes the outermost index, col4 becomes inner index. Values of col3, col4 become the index values.

* "reset_index" does the opposite of "set_index", the hierarchical index are moved into columns.

‡ By default, 'col3' and 'col4' will be removed from the DF, though you can leave them by option : `'drop = False'`.

Missing Data

Python	`NaN = np.nan` (not a number)
Pandas *	`NaN` or python built-in `None` mean missing/NA values

* Use `pd.isnull()`, `pd.notnull()` or `series1/df1.isnull()` to detect missing data.

Filtering Out Missing Data

`dropna()` returns with ONLY non-null data, source data NOT modified.

`df1.dropna()`	# drop any row containing missing value
`df1.dropna(axis = 1)`	# drop any column containing missing values
`df1.dropna(how = 'all')`	# drop row that are all missing
`df1.dropna(thresh = 3)`	# drop any row containing < 3 number of observations

Filling in Missing Data

`df2 = df1.fillna(0)`	# fill all missing data with 0
`df1.fillna('inplace = True')`	# modify in-place
Use a different fill value for each column : `df1.fillna({'col1' : 0, 'col2' : -1})`	
Only forward fill the 2 missing values in front : `df1.fillna(method = 'ffill', limit = 2)` i.e. for column1, if row 3-6 are missing. so 3 and 4 get filled with the value from 2, NOT 5 and 6.	